Parallel Programming Clusters with MPI

Ramses van Zon

SciNet HPC Consortium

June 11-12, 2018

Outline

- Distributed Memory Computing
- MPI: Basics
- MPI: Send & Receive
- MPI: Collectives
- Example: 1D Diffusion
- MPI: Performance/Scaling
- MPI: Non-Blocking Communications
- MPI: MPI-IO

Distributed Memory Computing

HPC Systems

Architectures

- Clusters, or, distributed memory machines
  - A bunch of servers linked together by a network ("interconnect").
  - GigE, Infiniband, Cray Gemini/Aries, IBM BGQ Torus
- Symmetric Multiprocessor (SMP) machines, or, shared memory machines
  - These can all see the same memory, typically a limited number of cores.
  - Present in virtually all systems these days.
- Vector machines
  - No longer dominant in HPC anymore.
  - Cray, NEC
- Accelerator (GPU, Cell, MIC, FPGA)
  - Heterogeneous use of standard CPU's with a specialized accelerator.
  - NVIDIA, AMD, Intel, Xilinx, Altera

Distributed Memory: Clusters

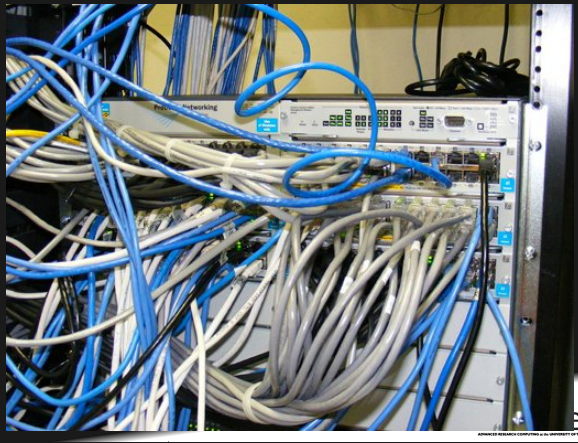
Simplest type of parallel computer to build

- Take existing powerful standalone computers

Distributed Memory: Clusters

Simplest type of parallel computer to build

- Take existing powerful standalone computers
- And network them



Distributed Memory: Clusters

Each node is independent!

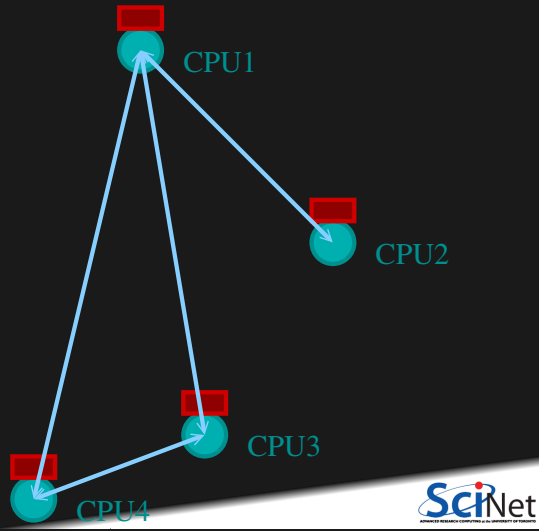
Parallel code consists of programs running on separate computers, communicating with each other.

Could be entirely different programs.

Each node has own memory!

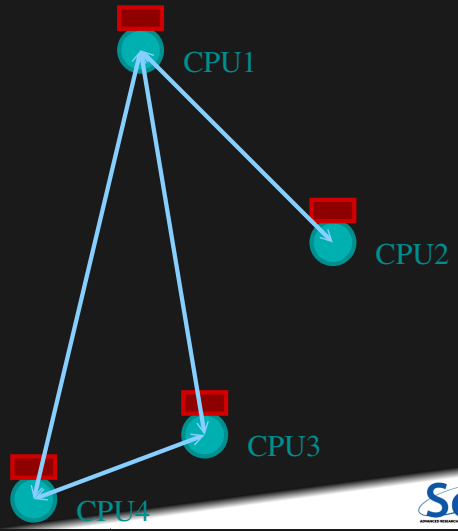
Whenever it needs data from another region, requests it from that CPU.

Usual model: "message passing"



Clusters+Message Passing

Hardware:
Easy to build
(Harder to build well)
Can build larger and larger clusters relatively easily
Software:
Every communication has to be hand-coded: hard to program



HPC Programming Models

Languages

- serial
  - C, C++, Fortran
- threaded (shared memory)
  - OpenMP, pthreads
- message passing (distributed memory)
  - MPI, PGAS (UPC, Coarray Fortran)
- accelerator (GPU, Cell, MIC, FPGA)
  - CUDA, OpenCL, OpenACC

Task (function, control) Parallelism

Work to be done is decomposed across processors

- e.g. divide and conquer
- each processor responsible for some part of the algorithm
- communication mechanism is significant
- must be possible for different processors to be performing different tasks

MPI: Basics

Message Passing Interface (MPI)

What is it?

- An open standard library interface for message passing, ratified by the MPI Forum
- Version: 1.0 (1994), 1.1 (1995), 1.2 (1997), 1.3 (2008)
- Version: 2.0 (1997), 2.1 (2008), 2.2 (2009)
- Version: 3.0 (2012), 3.1 (2015)

MPI Implementations

- OpenMPI `www.open-mpi.org`; up to version 3.0.0 now
  - Niagara: `module load gcc openmpi`
  - or: `module load intel openmpi`
- MPICH2 `www.mpich.org`
  - MPICH 3.x, MVAPICH2 2.x , IntelMPI 2018
  - Niagara: module load intel intelmpi

MPI is a Library for Message-Passing

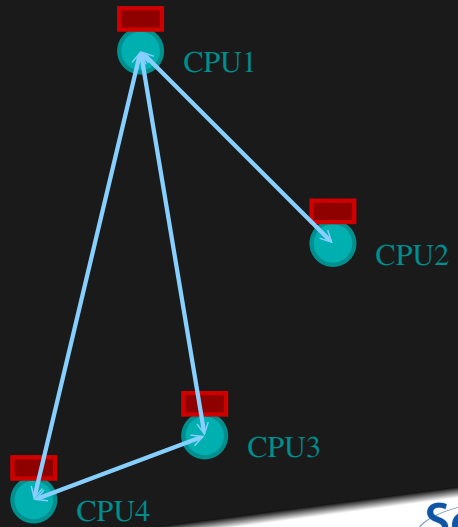
- Not built-in to compiler.
- Function calls that can be made from any compiler, many languages.
- Just link to it.
- Wrappers: mpicc, mpif90, mpicxx

```c
#include <stdio.h>
#include <mpi.h>
int main(int argc, char **argv) {
  int rank, size, err;
  err = MPI_Init(&argc, &argv);
  err = MPI_Comm_size(MPI_COMM_WORLD, &size);
  err = MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  printf("Hello world from task %d of %d!\n")
  err = MPI_Finalize();
}
```

```fortran
program helloworld
use mpi
implicit none
integer :: rank, commsize, err
call MPI_Init(err)
call MPI_Comm_size(MPI_COMM_WORLD, comsize, err)
call MPI_Comm_size(MPI_COMM_WORLD, rank, err)
print *,'Hello world from task',rank,'of',commsize
call MPI_Finalize(err)
end program helloworld
```

MPI is a Library for Message-Passing

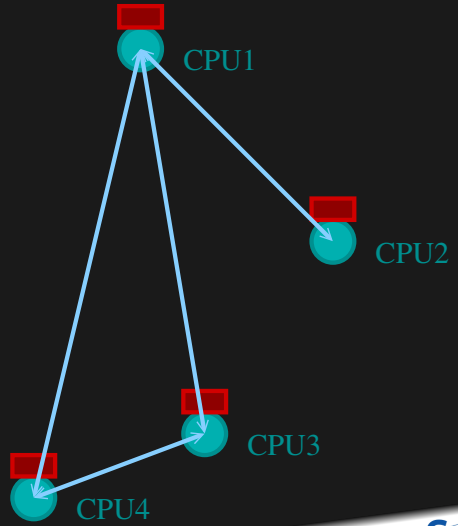
- Communication/coordination between tasks done by sending and receiving messages.
- Each message involves a function call from each of the programs.



MPI is a Library for Message-Passing

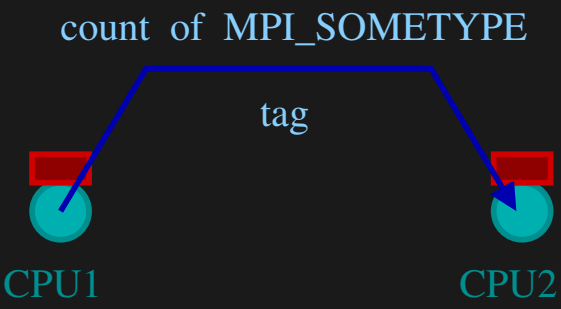
Three basic sets of functionality:

- Pairwise communications via messages
- Collective operations via messages
- Efficient routines for getting data from memory into messages and vice versa



Messages

- Messages have a sender and a receiver
- When you are sending a message, don't need to specify sender (it's the current processor),
- A sent message has to be actively received by the receiving process
- MPI messages are a string of length `count` all of some fixed MPI `type`
- MPI types exist for characters, integers, floating point numbers, etc.
- An arbitrary non-negative integer **tag** is also included – helps keep things straight if lots of messages are sent.



Size of MPI Library

- Many, many functions (>200)
- Not nearly so many concepts
- We'll get started with just 10-12, use more as needed.

```
MPI_Init()

MPI_Comm_size()

MPI_Comm_rank()

MPI_Ssend()

MPI_Recv()

MPI_Finalize()
```

Access to the Niagara supercomputer

Access to Niagara's Test Development System

- Log into Niagara with your *Compute Canada* account or your *scinetguestNNN* account.
- Proceed to go to the *Test Development System*, which is the part of Niagara we will for many of the summer school sessions.

```
$ ssh -Y USER@niagara.computecanada.ca
$ tds
$ cd $SCRATCH
$ cp -r /bb/scinet/course/ss2018/1_hpc/2_mpi .
$ cd 2_mpi
$ source setup
```

Access to the Niagara supercomputer

Access to Niagara's Test Development System

- Log into Niagara with your *Compute Canada* account or your *scinetguestNNN* account.
- Proceed to go to the *Test Development System*, which is the part of Niagara we will for many of the summer school sessions.

```
$ ssh -Y USER@niagara.computecanada.ca
$ tds
$ cd $SCRATCH
$ cp -r /bb/scinet/course/ss2018/1_hpc/2_mpi .
$ cd 2_mpi
$ source setup
```

Running computations

- On most supercomputer, a scheduler governs the allocation of resources.
- This means submitting a job with a jobscript.
- `srun`: a command that is a resource request + job running command all in one, and will run the command on one (or more) of the available resources.
- We have set aside 80-120 cores for the summer school, so occasionally, in busy sessions, you may have to wait for someone else's `srun` command to finish.

Example: Hello World

- The obligatory starting point
- `cd 2_mpi/mpi-intro`
- Compile and run it together

C:

```c
#include <stdio.h>
#include <mpi.h>
int main(int argc, char **argv) {
  int rank, size;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  printf("Hello world from task %d of %d!\n")
  MPI_Finalize();
}
```

Fortran:

```fortran
program helloworld
use mpi
implicit none
integer :: rank, commsize, err
call MPI_Init(err)
call MPI_Comm_size(MPI_COMM_WORLD, comsize, err)
call MPI_Comm_size(MPI_COMM_WORLD, rank, err)
print *,'Hello world from task',rank,'of',commsize
call MPI_Finalize(err)
end program helloworld
```

Example: Hello World

- The obligatory starting point
- `cd 2_mpi/mpi-intro`
- Compile and run it together

C:

```c
#include <stdio.h>
#include <mpi.h>
int main(int argc, char **argv) {
  int rank, size;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  printf("Hello world from task %d of %d!\n")
  MPI_Finalize();
}
```

Fortran:

```fortran
program helloworld
use mpi
implicit none
integer :: rank, commsize, err
call MPI_Init(err)
call MPI_Comm_size(MPI_COMM_WORLD, comsize, err)
call MPI_Comm_size(MPI_COMM_WORLD, rank, err)
print *,'Hello world from task',rank,'of',commsize
call MPI_Finalize(err)
end program helloworld
```

```
$ source $SCRATCH/1_hpc/2_mpi/setup
$ mpif90 hello-world.f90 -o hello-worldf
or
$ mpicc hello-world.c -o hello-worldc
$ srun -n 1 hello-world
$ srun -n 2 hello-world
$ srun -n 8 hello-world
```

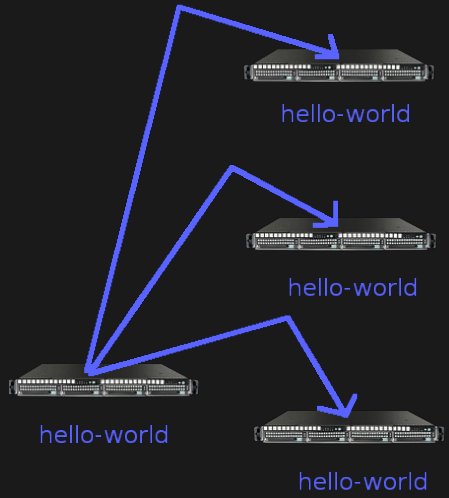
What does mpicc/mpif77 do?

- Just wrappers for the regular C, Fortran compilers that have the various `-I`, `-L` clauses in there automaticaly.
- `--showme` (OpenMPI) shows which options are being used.

```
$ mpicc --showme hello-world.c -o hello-worldc
gcc hello-world.c -o hello-world -I/scinet/niagara/software/2018a/opt/gcc-7.3.0/openmpi/3.1.0/include/ope
-I/scinet/niagara/software/2018a/opt/gcc-7.3.0/openmpi/3.1.0/include/openmpi/opal/mca/hwloc/hwloc1117/hwl
-I/scinet/niagara/software/2018a/opt/gcc-7.3.0/openmpi/3.1.0/include/openmpi/opal/mca/event/libevent2022/
-I/scinet/niagara/software/2018a/opt/gcc-7.3.0/openmpi/3.1.0/include/openmpi/opal/mca/event/libevent2022/
-I/scinet/niagara/software/2018a/opt/gcc-7.3.0/openmpi/3.1.0/include -pthread -L/opt/slurm/lib64
-L/scinet/niagara/mellanox/hpcx-2.1.0-ofed-4.3/hcoll/lib -L/scinet/niagara/mellanox/hpcx-2.1.0-ofed-4.3/m
-L/scinet/niagara/mellanox/hpcx-2.1.0-ofed-4.3/ucx/lib -Wl,-rpath -Wl,/opt/slurm/lib64 -Wl,-rpath
-Wl,/scinet/niagara/mellanox/hpcx-2.1.0-ofed-4.3/hcoll/lib -Wl,-rpath
-Wl,/scinet/niagara/mellanox/hpcx-2.1.0-ofed-4.3/mxm/lib -Wl,-rpath
-Wl,/scinet/niagara/mellanox/hpcx-2.1.0-ofed-4.3/ucx/lib -Wl,-rpath
-Wl,/scinet/niagara/software/2018a/opt/gcc-7.3.0/openmpi/3.1.0/lib -Wl,--enable-new-dtags
-L/scinet/niagara/software/2018a/opt/gcc-7.3.0/openmpi/3.1.0/lib -lmpi
$
```

What mpirun/srun does

- Launches *n* processes, assigns each an MPI **rank** and starts the program
- For multinode run, has a list of nodes, ssh's to each node and launches the program
- `mpirun` only runs the processes on the login node, and does not allocate resources; typically used inside a batch job.
- `srun` allocates the resources on the cluster and runs the processes there: This is what we'll use in the summer school.



Number of Processes

- Number of processes to use is almost always equal to the number of processors on a node.
- But not necessarily.
- If hyperthreading: multiple processes per core (not enabled in the TDS scheduler).
- If memory-hungry: less processes than cores on a node (for Niagara, if $>$ 4GB/process).
- If hybrid (threaded+mpi): less processes per core, but multiple threads per core, usual one thread per core.

Number of Processes

- Number of processes to use is almost always equal to the number of processors on a node.
- But not necessarily.
- If hyperthreading: multiple processes per core (not enabled in the TDS scheduler).
- If memory-hungry: less processes than cores on a node (for Niagara, if $>$ 4GB/process).
- If hybrid (threaded+mpi): less processes per core, but multiple threads per core, usual one thread per core.

Regular pure mpi run on a 40 core node:

```
$ srun -N 1 -n 40 hello-worldc
```

Number of Processes

- Number of processes to use is almost always equal to the number of processors on a node.
- But not necessarily.
- If hyperthreading: multiple processes per core (not enabled in the TDS scheduler).
- If memory-hungry: less processes than cores on a node (for Niagara, if $>$ 4GB/process).
- If hybrid (threaded+mpi): less processes per core, but multiple threads per core, usual one thread per core.

Regular pure mpi run on a 40 core node:

```
$ srun -N 1 -n 40 hello-worldc
```

Hyperthreaded mpi run (not on TDS):

```
$ srun -N 1 -n 80 hello-worldc
```

Number of Processes

- Number of processes to use is almost always equal to the number of processors on a node.
- But not necessarily.
- If hyperthreading: multiple processes per core (not enabled in the TDS scheduler).
- If memory-hungry: less processes than cores on a node (for Niagara, if $>$ 4GB/process).
- If hybrid (threaded+mpi): less processes per core, but multiple threads per core, usual one thread per core.

Regular pure mpi run on a 40 core node:

```
$ srun -N 1 -n 40 hello-worldc
```

Hyperthreaded mpi run (not on TDS):

```
$ srun -N 1 -n 80 hello-worldc
```

Memory-hungry mpi run on a 40 core node requiring 8GB per process:

```
$ srun -N 1 -n 20 hello-worldc
```

Number of Processes

- Number of processes to use is almost always equal to the number of processors on a node.
- But not necessarily.
- If hyperthreading: multiple processes per core (not enabled in the TDS scheduler).
- If memory-hungry: less processes than cores on a node (for Niagara, if $>$ 4GB/process).
- If hybrid (threaded+mpi): less processes per core, but multiple threads per core, usual one thread per core.

Regular pure mpi run on a 40 core node:

```
$ srun -N 1 -n 40 hello-worldc
```

Hyperthreaded mpi run (not on TDS):

```
$ srun -N 1 -n 80 hello-worldc
```

Memory-hungry mpi run on a 40 core node requiring 8GB per process:

```
$ srun -N 1 -n 20 hello-worldc
```

Hybrid run (8 mpi processes with 5 threads):

```
$ srun -N 1 -n 8 -c 5 hello-worldc
```

Number of Processes

- Number of processes to use is almost always equal to the number of processors on a node.
- But not necessarily.
- If hyperthreading: multiple processes per core (not enabled in the TDS scheduler).
- If memory-hungry: less processes than cores on a node (for Niagara, if $>$ 4GB/process).
- If hybrid (threaded+mpi): less processes per core, but multiple threads per core, usual one thread per core.

In this session, omit the `-N` argument and use `srun` with a `-n` argument only.

Regular pure mpi run on a 40 core node:

```
$ srun -N 1 -n 40 hello-worldc
```

Hyperthreaded mpi run (not on TDS):

```
$ srun -N 1 -n 80 hello-worldc
```

Memory-hungry mpi run on a 40 core node requiring 8GB per process:

```
$ srun -N 1 -n 20 hello-worldc
```

Hybrid run (8 mpi processes with 5 threads):

```
$ srun -N 1 -n 8 -c 5 hello-worldc
```

mpirun / srun runs any program

- mpirun will start that process launching procedure for any program
- Sets variables somehow that mpi programs recognize so that they know which process they are.

```
$ hostname
tds01.scinet.local
$ mpirun -n 2 hostname
tds01.scinet.local
tds01.scinet.local
$ srun -n 2 hostname
tds02.scinet.local
tds02.scinet.local
$
```

Example: “Hello World”

```
$ srun -n 4 ./hello-worldc
Hello from task 2 of 4 world
Hello from task 1 of 4 world
Hello from task 0 of 4 world
Hello from task 3 of 4 world
```

```
$ srun --label -n 4 ./hello-worldc
2: Hello from task 2 of 4 world
1: Hello from task 1 of 4 world
0: Hello from task 0 of 4 world
4: Hello from task 3 of 4 world
```

Make

- Make builds an executable from a list of source code files and rules
- Many files to do, of which order doesn't matter for most
- Parallelism!
- `make -j N` launches N processes to do it.

```
$ make
$ make -j 2
$ make -j
```

What the code does (Fortran)

```
program helloworld
use mpi
implicit none
integer :: rank, commsize, err

call MPI_Init(err)
call MPI_Comm_size(MPI_COMM_WORLD, comsize, err)
call MPI_Comm_size(MPI_COMM_WORLD, rank, err)

print *,'Hello world from task',rank,'of',commsize

call MPI_Finalize(err)
end program helloworld
```

- `use mpi`: imports declarations for MPI function calls
- `call MPI_INIT(err)`: initialization for MPI library. Must come first.
- `err`: Returns any error code.
- `call MPI_FINALIZE(err)`: close up MPI stuff. Must come last. `err`: Returns any error code.
- `call MPI_COMM_RANK, call MPI_COMM_SIZE`: requires a little more exposition.

What the code does (C)

- `#include <mpi.h>` - MPI library definitions
- `MPI_Init(&argc,&argv)`
  MPI Intialization, must come first
- `MPI_Finalize()`
  Finalizes MPI, must come last
- `err` - MPI routine could return an error code

```c
#include <stdio.h>
#include <mpi.h>

int main(int argc, char **argv) {

 int rank, size;
 int err;

 err = MPI_Init(&argc, &argv);

 err = MPI_Comm_size(MPI_COMM_WORLD, &size);
 err = MPI_Comm_rank(MPI_COMM_WORLD, &rank);

 printf("Hello, world from task %d of %d!\n",rank,si

 MPI_Finalize();

}
```

What the code does (C)

- `#include <mpi.h>` - MPI library definitions
- `MPI_Init(&argc,&argv)`
  MPI Intialization, must come first
- `MPI_Finalize()`
  Finalizes MPI, must come last
- `err` - MPI routine could return an error code

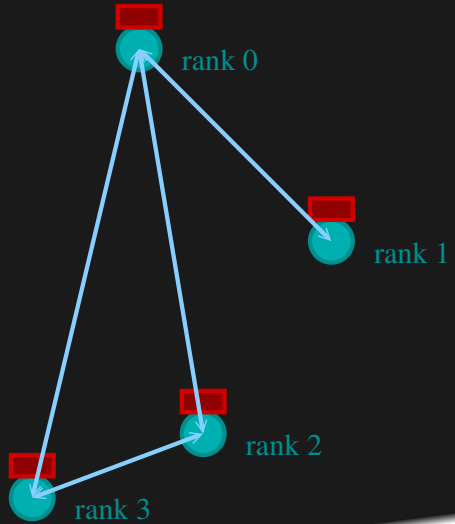
Communicator Components

- A communicator is a handle to a group of processes that can communicate.
- `MPI_Comm_rank(MPI_COMM_WORLD,&rank)`
- `MPI_Comm_size(MPI_COMM_WORLD,&rank)`

```c
#include <stdio.h>
#include <mpi.h>

int main(int argc, char **argv) {

 int rank, size;
 int err;

 err = MPI_Init(&argc, &argv);

 err = MPI_Comm_size(MPI_COMM_WORLD, &size);
 err = MPI_Comm_rank(MPI_COMM_WORLD, &rank);

 printf("Hello, world from task %d of %d!\n",rank,si

 MPI_Finalize();

}
```

Communicators

- MPI groups processes into communicators.
- Each communicator has some size – number of tasks.
- Every task has a rank 0..size-1
- Every task in your program belongs to `MPI_COMM_WORLD`.



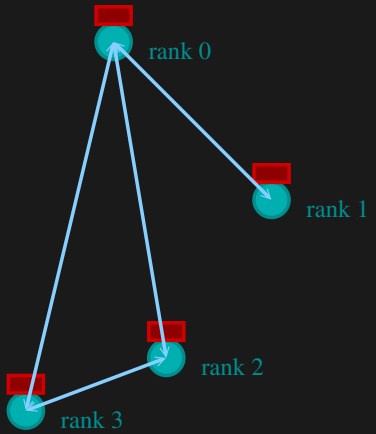
`MPI_COMM_WORLD`:
size = 4, ranks = 0..3

Communicators

- One can create one's own communicators over the same tasks.
- May break the tasks up into subgroups.
- May just re-order them for some reason

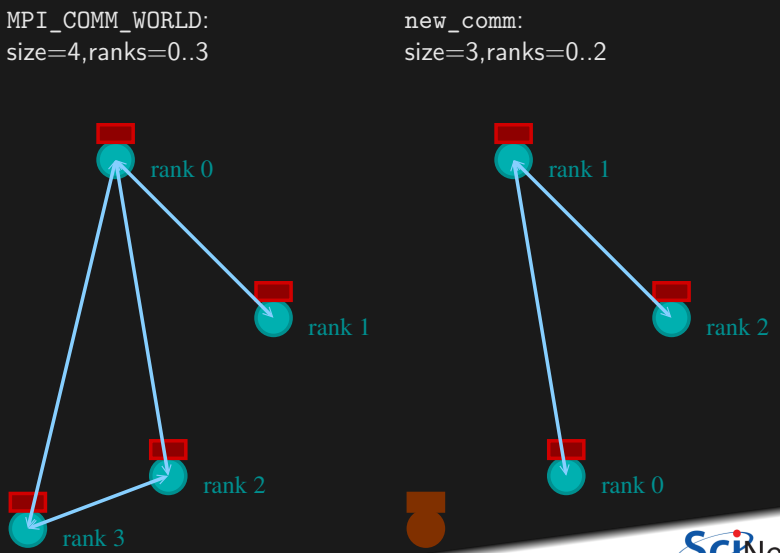
Communicators

- One can create one's own communicators over the same tasks.
- May break the tasks up into subgroups.
- May just re-order them for some reason



Communicators

- One can create one's own communicators over the same tasks.
- May break the tasks up into subgroups.
- May just re-order them for some reason



MPI Communicator Basics

Communicator Components

- `MPI_COMM_WORLD`:
  Global Communicator
- `MPI_Comm_rank(MPI_COMM_WORLD,&rank)`
  Get current tasks rank
- `MPI_Comm_size(MPI_COMM_WORLD,&size)`
  Get communicator size

Send & Receive

MPI: Send & Receive

`hello-world` was our first real MPI program
But no Messages were being Passed.

- Let's fix this

MPI: Send & Receive

`hello-world` was our first real MPI program
But no Messages were being Passed.

- Let's fix this
- `mpicc -o firstmessagec firstmessage.c`
- `srun -n 2 ./firstmessagec`
- Note: C - MPI_CHAR

```c
#include <stdio.h>
#include <mpi.h>
int main(int argc, char **argv) {
  int rank, size;
  int sendto, recvfrom;  /*task to send,recv from*/
  int ourtag=1;          /*tag to label msgs*/
  char sendmsg[]="Hello";/*text to send*/
  char getmsg[6];        /*text to receive*/
  MPI_Status rstatus;    /*recv status info*/
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  if (rank == 0) {
    sendto = 1;
    MPI_Ssend(sendmsg, 6, MPI_CHAR, sendto,
              ourtag, MPI_COMM_WORLD);
    printf("%d: Sent msg <%s>\n",rank,sendmsg);
  } else if (rank == 1) {
    recvfrom = 0;
    MPI_Recv(getmsg, 6, MPI_CHAR, recvfrom,
             ourtag, MPI_COMM_WORLD, &rstatus);
    printf("%d: Got msg <%s>\n", rank, getmsg);
  }
  MPI_Finalize();
}
```

MPI: Send & Receive

- Let's fix this, Fortran version

MPI: Send & Receive

- Let's fix this, Fortran version
- `mpif90 -o firstmessagef firstmessage.f90`
- `srun -np 2 ./firstmessagef`
- Note Fortran: MPI_CHARACTER

```
program firstmessage
use mpi
implicit none
integer :: rank,comsize,err
integer :: sendto,recvfrom  !Task to send,recv from
integer :: ourtag=1         !tag to label msgs
character(5) :: sendmessage !text to send
character(5) :: getmessage  !text rcvd
integer, dimension(MPI_STATUS_SIZE) :: rstatus
call MPI_Init(err)
call MPI_Comm_rank(MPI_COMM_WORLD, rank, err)
call MPI_Comm_size(MPI_COMM_WORLD, comsize, err)
if (rank == 0) then
 sendmessage = 'Hello'
 sendto = 1
 call MPI_Ssend(sendmessage,5,MPI_CHARACTER,sendto,&
                ourtag,MPI_COMM_WORLD,err)
 print *, rank, ' sent message <',sendmessage,'>'
else if (rank == 1) then
 recvfrom = 0
 call MPI_Recv(getmessage,5,MPI_CHARACTER,recvfrom,&
               ourtag,MPI_COMM_WORLD,rstatus,err)
 print *, rank, ' got message <',getmessage,'>'
endif
call MPI_Finalize(err)
end program firstmessage
```

Send and Receive

C

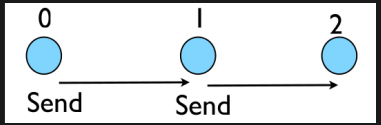
```
MPI_Status status;
err = MPI_Ssend(sendptr, count, MPI_TYPE, destination, tag, Communicator);
err = MPI_Recv(rcvptr, count, MPI_TYPE, source, tag, Communicator, status);
```

Fortran

```
integer status(MPI_STATUS_SIZE)
call MPI_SSEND(sendarr, count, MPI_TYPE, destination, tag, Communicator, err)
call MPI_RECV(rcvarr, count, MPI_TYPE, source, tag, Communicator, status, err)
```

More Complicated Example

Send a message to the right:



Specials

Special Source/Destination `MPI_PROC_NULL`

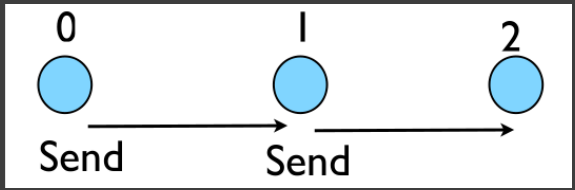
`MPI_PROC_NULL` basically ignores the relevant operation; can lead to cleaner code.

Special Source `MPI_ANY_SOURCE`

`MPI_ANY_SOURCE` is a wildcard; matches any source when receiving.

MPI: Send Right, Receive Left

```
#include <iostream>
#include <string>
#include <mpi.h>
using namespace std;
int main(int argc, char **argv) {
    int         rank, size, err, left, right, tag = 1;
    double      msgsent, msgrcvd;
    MPI_Status  rstatus;
    err = MPI_Init(&argc, &argv);
    err = MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    err = MPI_Comm_size(MPI_COMM_WORLD, &size);
    left = rank - 1;
    if (left < 0) left = MPI_PROC_NULL;
    right = rank + 1;
    if (right >= size) right = MPI_PROC_NULL;
    msgsent = rank*rank;
    msgrcvd = -999.;
    err = MPI_Ssend(&msgsent, 1, MPI_DOUBLE, right, tag, MPI_COMM_WORLD);
    err = MPI_Recv(&msgrcvd, 1, MPI_DOUBLE, left, tag, MPI_COMM_WORLD, &rstatus);
    cout << to_string(rank) + ": Sent " + to_string(msgsent) + " and got " + to_string(msgrcvd) + "\n";
    err = MPI_Finalize();
}
```



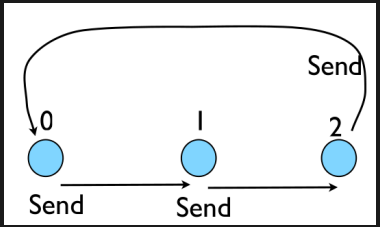
MPI: Send Right, Receive Left

```
$ make secondmessagec
$ srun -n 3 ./secondmessagec
2: Sent 4.000000 and got 1.000000
0: Sent 0.000000 and got -999.000000
1: Sent 1.000000 and got 0.000000
$
```

```
$ srun -n 6 ./secondmessagec
4: Sent 16.000000 and got 9.000000
5: Sent 25.000000 and got 16.000000
0: Sent 0.000000 and got -999.000000
1: Sent 1.000000 and got 0.000000
2: Sent 4.000000 and got 1.000000
3: Sent 9.000000 and got 4.000000
```

MPI: Send Right, Receive Left with Periodic BCs

Periodic Boundary Conditions:



MPI: Send Right, Receive Left with Periodic BCs

```
    ...
    left = rank - 1;
    if (left < 0) left = size-1; // Periodic BC
    right = rank + 1;
    if (right >= size) right =0; // Periodic BC
    msgsent = rank*rank;
    msgrcvd = -999.;
    ...
```

```
$ make thirdmessagec # or thirdmessagef
$ srun -n 5 thirdmessagec
```

MPI: Send Right, Receive Left with Periodic BCs

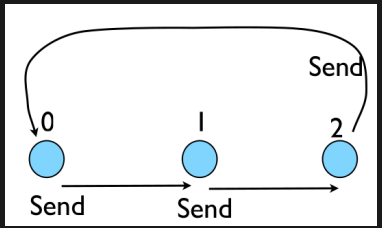
```
    ...
    left = rank - 1;
    if (left < 0) left = size-1; // Periodic BC
    right = rank + 1;
    if (right >= size) right =0; // Periodic BC
    msgsent = rank*rank;
    msgrcvd = -999.;
    ...
```

```
$ make thirdmessagec # or thirdmessagef
$ srun -n 5 thirdmessagec
```

Just sort of hangs there doing nothing?

Deadlock!

- A classic parallel bug.
- Occurs when a cycle of tasks are waiting for the others to finish.
- Whenever you see a closed cycle, you likely have (or risk) a deadlock.
- Here, all processes are waiting for the send to complete, but no one is receiving.



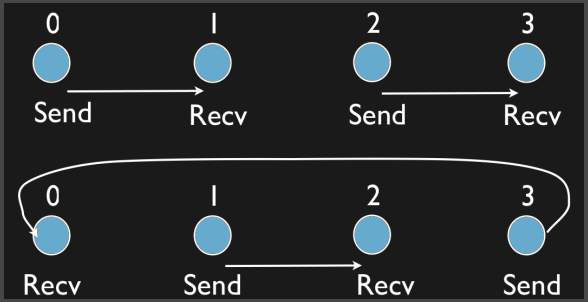
Big MPI Lesson #1

All sends and receives must be paired at the time of sending

How do we fix the deadlock?

Without using new MPI routine, how do we fix the deadlock?

Even-odd solution



- First: evens send, odds receive
- Then: odds send, evens receive
- Will this work with an odd number of processes? How about 2? 1?

MPI: Send Right, Receive Left with Periodic BCs - fixed

```
...
if ((rank % 2) == 0) {
    err =  MPI_Ssend(&msgsent, 1, MPI_DOUBLE, right, tag, MPI_COMM_WORLD);
    err = MPI_Recv(&msgrcvd, 1, MPI_DOUBLE, left, tag, MPI_COMM_WORLD, &rstatus);
} else {
    err = MPI_Recv(&msgrcvd, 1, MPI_DOUBLE, left, tag, MPI_COMM_WORLD, &rstatus);
    err =  MPI_Ssend(&msgsent, 1, MPI_DOUBLE, right, tag, MPI_COMM_WORLD);
}
...
```

MPI: Send Right, Receive Left with Periodic BCs - fixed

```
    ...
    if ((rank % 2) == 0) {
        err =  MPI_Ssend(&msgsent, 1, MPI_DOUBLE, right, tag, MPI_COMM_WORLD);
        err = MPI_Recv(&msgrcvd, 1, MPI_DOUBLE, left, tag, MPI_COMM_WORLD, &rstatus);
    } else {
        err = MPI_Recv(&msgrcvd, 1, MPI_DOUBLE, left, tag, MPI_COMM_WORLD, &rstatus);
        err =  MPI_Ssend(&msgsent, 1, MPI_DOUBLE, right, tag, MPI_COMM_WORLD);
    }
    ...
```

```
$ make fourthmessagec
$ srun -n 5 ./fourthmessagec
1: Sent 1.000000 and got 0.000000
2: Sent 4.000000 and got 1.000000
3: Sent 9.000000 and got 4.000000
4: Sent 16.000000 and got 9.000000
0: Sent 0.000000 and got 16.000000
```

MPI: Sendrecv

```
err = MPI_Sendrecv(sendptr, count, MPI_TYPE, destination, tag,
                   recvptr, count, MPI_TYPE, source, tag, Communicator, MPI_Status)
```

- A blocking send and receive built together
- Lets them happen simultaneously
- Can automatically pair send/recvs
- Why 2 sets of tags/types/counts?

Send Right, Receive Left with Periodic BCs - Sendrecv

Code

```
    ...
    err =  MPI_Sendrecv(&msgsent, 1, MPI_DOUBLE, right, tag,
             &msgrcvd, 1, MPI_DOUBLE, left, tag, MPI_COMM_WORLD, &rstatus);
    ...
```

Execution

```
$ make fifthmessage
$ srun -n 5 ./fifthmessage
1: Sent 1.000000 and got 0.000000
2: Sent 4.000000 and got 1.000000
3: Sent 9.000000 and got 4.000000
4: Sent 16.000000 and got 9.000000
0: Sent 0.000000 and got 16.000000
```

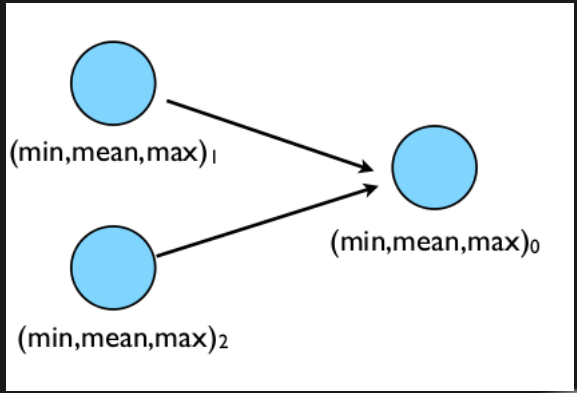
Different versions of SEND

To DO

Collectives

Reductions: Min, Mean, Max Example

- Calculate the min/mean/max of random numbers -1.0 ... 1.0
- Should trend to -1/0/+1 for a large N.
- How to MPI it?
- Partial results on each node, collect all to node 0.



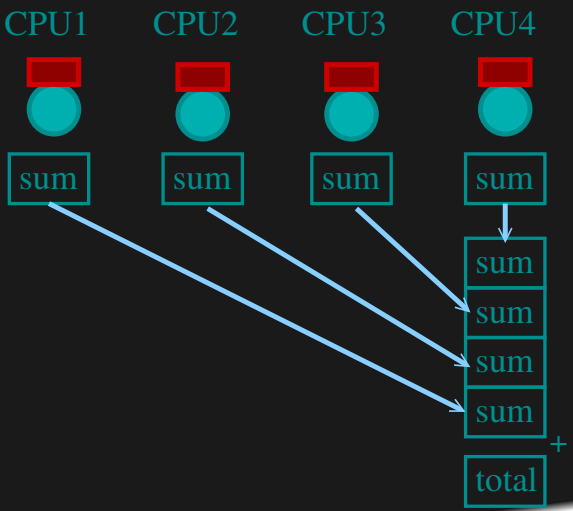
Reductions: Min, Mean, Max Example

```
#include <mpi.h>
#include <iostream>
#include <algorithm>
#include <cstdlib>
using namespace std;
int main(int argc, char **argv) {
   const int nx = 1500, MIN=0, MEAN=1, MAX=2;
   double mmm[3] = {1e+19, 0, -1e+19};
   int rank, size, tag = 1;
   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   double *dat = new double[nx];
   srand(0);
   for (int i=0;i<dx*rank;i++) rand();
   for (int i=0;i<nx;i++)
      dat[i] = 2*((double)rand()/RAND_MAX)-1.;
   for (int i=0;i<nx;i++) {
      mmm[MIN] = min(dat[i], mmm[MIN]);
      mmm[MAX] = max(dat[i], mmm[MAX]);
      mmm[MEAN] += dat[i];
   }
   mmm[MEAN] /= nx;
   if (rank != 0)
      MPI_Ssend(mmm, 3, MPI_DOUBLE, 0, tag,
                MPI_COMM_WORLD);
   else {
      double recvmmm[3];
      for (int i=1;i<size;i++) {
         MPI_Recv(recvmmm, 3, MPI_DOUBLE,
                  MPI_ANY_SOURCE, tag,
                  MPI_COMM_WORLD,MPI_STATUS_IGNORE);
         mmm[MIN] = min(recvmmm[MIN], mmm[MIN]);
         mmm[MAX] = max(recvmmm[MAX], mmm[MAX]);
         mmm[MEAN] += recvmmm[MEAN];
      }
      mmm[MEAN] /= size;
      cout << "Global Min/mean/max " << mmm[MIN] <<
              globmmm[MEAN]<<" "<<mmm[MAX]<<endl;
   }
   MPI_Finalize();
}
```

Inefficient!

- Requires (P-1) messages
- 2(P-1) if everyone then needs to get the answer.

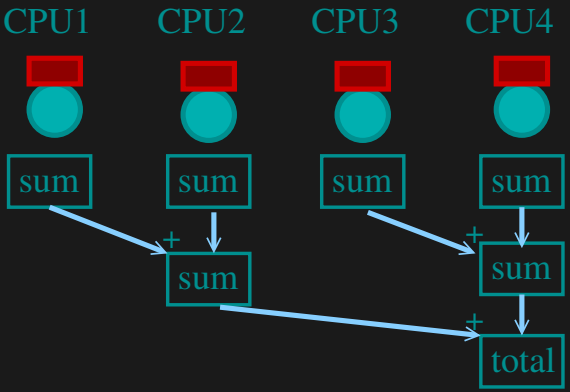
$$T_{comm} = PC_{comm}$$



Better Summing

- Pairs of processors; send partial sums
- Max messages received $\log_2(P)$
- Can repeat to send total back.

$$T_{comm} = 2\log_2(P)C_{comm}$$



Reduction: Works for a variety of operations (+,*,min,max)

MPI Collectives

```
err = MPI_Allreduce(sendptr, rcvptr, count, MPI_TYPE, MPI_Op, Communicator);
err = MPI_Reduce(sendbuf, recvbuf, count, MPI_TYPE, MPI_Op, root, Communicator);
```

- `sendptr/rcvptr`: pointers to buffers
- `count`: number of elements in ptrs
- `MPI_TYPE`: one of `MPI_DOUBLE`, `MPI_FLOAT`, `MPI_INT`, `MPI_CHAR`, etc.
- `MPI_Op`: one of `MPI_SUM`, `MPI_PROD`, `MPI_MIN`, `MPI_MAX`.
- `Communicator`: MPI_COMM_WORLD or user created.
- `All` variant send result back to all processes; non-`All` sends to process `root`.

Reductions: Min, Mean, Max with MPI Collectives

```
double globalmmm[3];
MPI_Allreduce(&mmm[MIN], &globalmmm[MIN], 1, MPI_DOUBLE, MPI_MIN, MPI_COMM_WORLD);
MPI_Allreduce(&mmm[MAX], &globalmmm[MAX], 1, MPI_DOUBLE, MPI_MAX, MPI_COMM_WORLD);
MPI_Allreduce(&mmm[MEAN], &globalmmm[MEAN], 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
globalmmm[MEAN] /= size;
if (rank==0)
   cout << "Global Min/mean/max " << mmm[MIN] << " " <<
           globmmm[MEAN]<<" "<<mmm[MAX] << endl;
```

Collective Operations

Collective

- Reductions are an example of a ***collective*** operation.
- As opposed to the pairwise messages we've seen before
- All processes in the communicator must participate.
- Cannot proceed until all have participated.
- Don't necessarity know what's 'under the hood'.

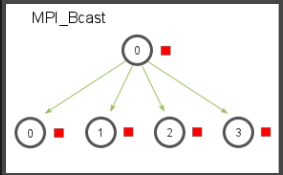
Collective Operations

Collective

- Reductions are an example of a ***collective*** operation.
- As opposed to the pairwise messages we've seen before
- All processes in the communicator must participate.
- Cannot proceed until all have participated.
- Don't necessarity know what's 'under the hood'.

Other MPI Collectives

Broadcast



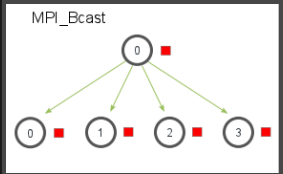
Collective Operations

Collective

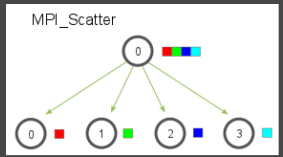
- Reductions are an example of a ***collective*** operation.
- As opposed to the pairwise messages we've seen before
- All processes in the communicator must participate.
- Cannot proceed until all have participated.
- Don't necessarity know what's 'under the hood'.

Other MPI Collectives

Broadcast



Scatter



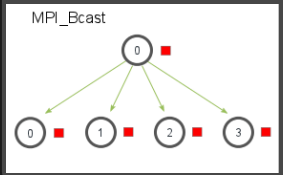
Collective Operations

Collective

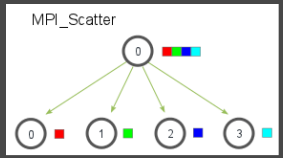
- Reductions are an example of a ***collective*** operation.
- As opposed to the pairwise messages we've seen before
- All processes in the communicator must participate.
- Cannot proceed until all have participated.
- Don't necessarity know what's 'under the hood'.

Other MPI Collectives

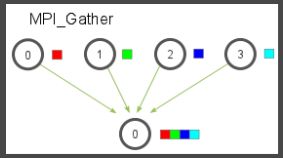
Broadcast



Scatter



Gather



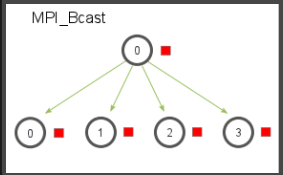
Collective Operations

Collective

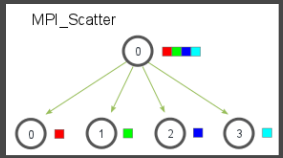
- Reductions are an example of a ***collective*** operation.
- As opposed to the pairwise messages we've seen before
- All processes in the communicator must participate.
- Cannot proceed until all have participated.
- Don't necessarity know what's 'under the hood'.

Other MPI Collectives

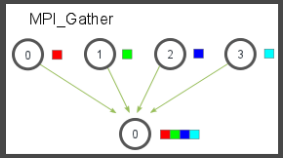
Broadcast



Scatter



Gather

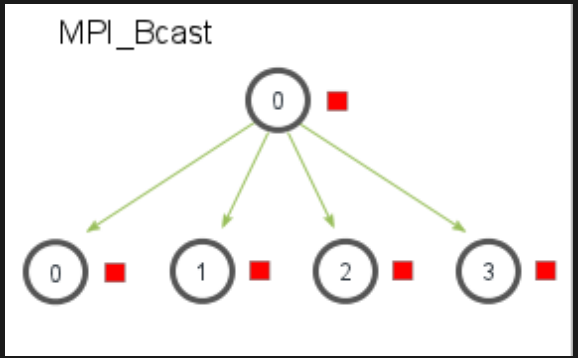


File I/O

Barriers (don't!)

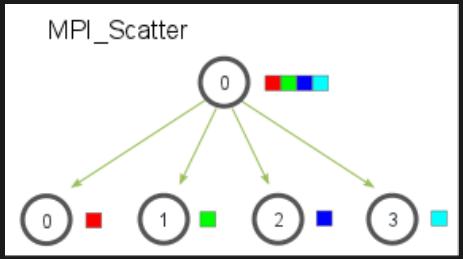
...

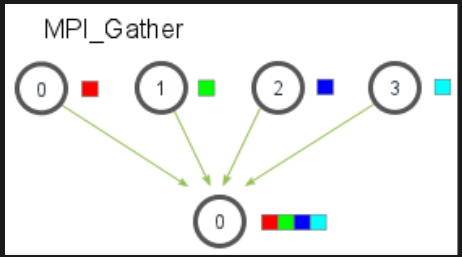
MPI_Collectives: Broadcast



- Broadcasts a message from process with rank "root" to all processes in group, including itself.
- Amount of data sent must be equal to amount of data received.
- `err = MPI_Bcast(void *buf, count, MPI_Type, root, Comm)`
  - `buf`: buffer of data to send/recv
  - `count`: number of elements in buf
  - `MPI_TYPE`: one of `MPI_DOUBLE`, `MPI_FLOAT`, `MPI_INT`, `MPI_CHAR`, etc.
  - `root`: "root" processor to send from
  - Communicator: MPI_COMM WORLD or user created

MPI_Collectives: Scatter/Gather





- Scatter: Sends data from "root" to all processes in group.
- `err = MPI_Scatter(void *send_buf, send_count, MPI_Type, void *recv_buf, recv_count, MPI_Type, root, Comm)`
- Gather: Recives data on "root" from all processes in group.
- `err = MPI_Gather(void *send_buf, send_count, MPI_Type, void *recv_buf, recv_count, MPI_Type, root, Comm)`

Example: Scatter/Gather

Scatter

Simple Scatter example sending data from root to 4 procesors.

```
$ cd $SCRATCH/2_mpi/collectives
$ make
$ srun -n 4 ./scatter
```

Gather

- Copy Scatter.c to Gather.c and reverse the process.
- Send from 4 processes and collect on root using `MPI_Gather()`.

MPI_Collectives: Barrier

- Blocks calling process until all group members have called it.
- Decreases performance. Try to avoid using it explicitly.
- `err = MPI_Barrier(Comm)`
  - Communicator Comm: `MPI_COMM_WORLD` or user created

MPI_Collectives: All-to-all

TO DO

Scientific MPI Example

Scientific MPI Examples

Real MPI Problems

- Finite Difference Stencils
- Time-Marching Method
- Domain Decomposition
- Load Balancing
- Global Norms
- Boundary Conditions

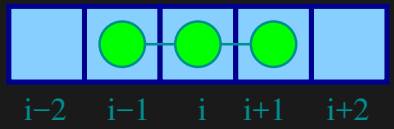
Discretizing Derivatives

- Partial Differential Equations like the diffusion equation

$$\frac{\partial T}{\partial t} = D\frac{\partial^2 T}{\partial x^2}$$

  are usually numerically solved by finite differencing the discretized values.
- Implicitly or explicitly involves interpolating data and taking the derivative of the interpolant.
- Larger 'stencils' $\longrightarrow$ More accuracy.

$$\frac{\partial^2 T}{\partial x^2} \approx \frac{T_{i+1} - 2T_i + T_{i-1}}{\Delta x^2}$$



Diffusion equation in higher dimensions

Spatial grid separation: Δx. Time step Δt.
Grid indices: i, j. Time step index: (n)

1D

$$\left.\frac{\partial T}{\partial t}\right|_i \approx \frac{T_i^{(n)} - T_i^{(n-1)}}{\Delta t}$$

$$\left.\frac{\partial^2 T}{\partial x^2}\right|_i \approx \frac{T_{i-1}^{(n)} - 2T_i^{(n)} + T_{i+1}^{(n)}}{\Delta x^2}$$

Diffusion equation in higher dimensions

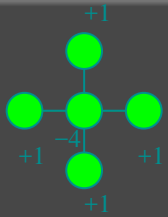
Spatial grid separation: Δx. Time step Δt.
Grid indices: i, j. Time step index: (n)

1D

$$\left.\frac{\partial T}{\partial t}\right|_i \approx \frac{T_i^{(n)} - T_i^{(n-1)}}{\Delta t}$$

$$\left.\frac{\partial^2 T}{\partial x^2}\right|_i \approx \frac{T_{i-1}^{(n)} - 2T_i^{(n)} + T_{i+1}^{(n)}}{\Delta x^2}$$

2D



Diffusion equation in higher dimensions

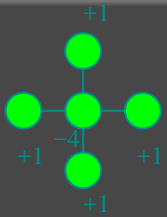
Spatial grid separation: Δx. Time step Δt.
Grid indices: i, j. Time step index: (n)

1D

$$\left.\frac{\partial T}{\partial t}\right|_i \approx \frac{T_i^{(n)} - T_i^{(n-1)}}{\Delta t}$$

$$\left.\frac{\partial^2 T}{\partial x^2}\right|_i \approx \frac{T_{i-1}^{(n)} - 2T_i^{(n)} + T_{i+1}^{(n)}}{\Delta x^2}$$

2D



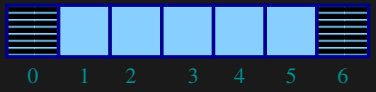
$$\left.\frac{\partial T}{\partial t}\right|_{i,j} \approx \frac{T_{i,j}^{(n)} - T_{i,j}^{(n-1)}}{\Delta t}$$

$$\left.\left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2}\right)\right|_{i,j} \approx \frac{T_{i-1,j}^{(n)} + T_{i,j-1}^{(n)} - 4T_{i,j}^{(n)} + T_{i+1,j}^{(n)} + T_{i,j+1}^{(n)}}{\Delta x^2}$$

Stencils and Boundaries

- How do you deal with boundaries?
- The stencil juts out, you need info on cells beyond those you're updating.
- Common solution: ***Guard cells***:
  - Pad domain with these guard celss so that stencil works even for the first point in domain.
  - Fill guard cells with values such that the required boundary conditions are met.

1D

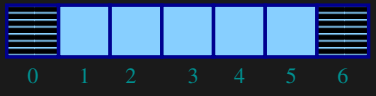


- Number of guard cells $n_g = 1$
- Loop from $i = n_g \ldots N - 2n_g$.

Stencils and Boundaries

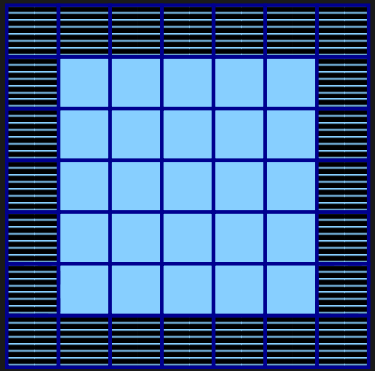
- How do you deal with boundaries?
- The stencil juts out, you need info on cells beyond those you're updating.
- Common solution: ***Guard cells***:
  - Pad domain with these guard celss so that stencil works even for the first point in domain.
  - Fill guard cells with values such that the required boundary conditions are met.

1D



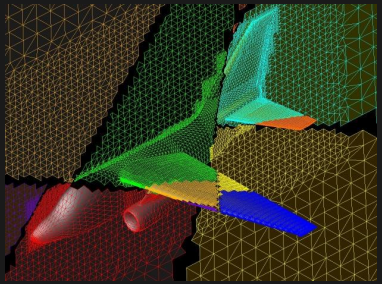
- Number of guard cells $n_g = 1$
- Loop from $i = n_g \ldots N - 2n_g$.

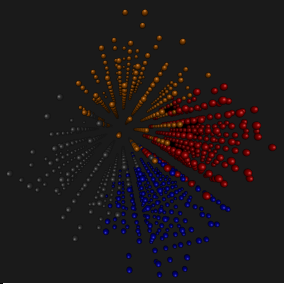
2D

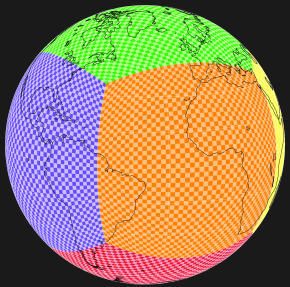


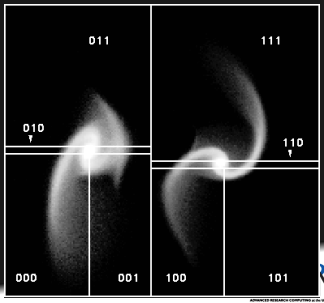
Domain decomposition

- A very common approach to parallelizing on distributed memory computers.
- Subdivide the domain into contiguous subdomains.
- Give each subdomain to a different MPI process.
- No process contains the full data!
- Maintains locality.
- Need mostly local data, ie., only data at the boundary of each subdomain will need to be sent between processes.



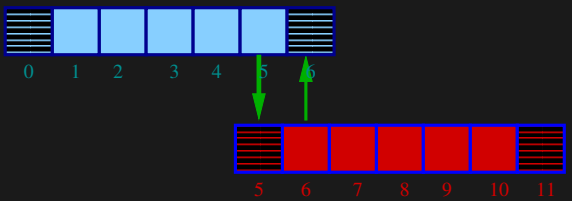






Guard cell exchange

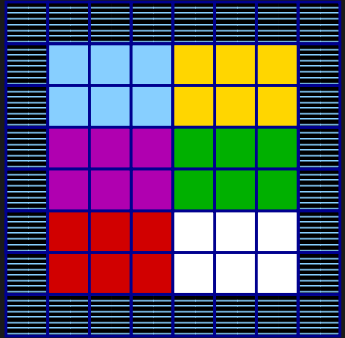
- In the domain decomposition, the stencils will jut out into a neighbouring subdomain.
- Much like the boundary condition.
- One uses guard cells for domain decomposition too.
- If we managed to fill the guard cell with values from neighbouring domains, we can treat each coupled subdomain as an isolated domain with changing boundary conditions.

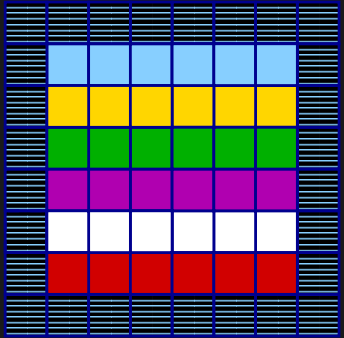


- Could use even/odd trick, or sendrecv.

2D diffusion with MPI

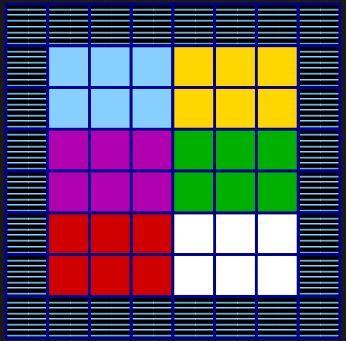
How to divide the work in 2d?

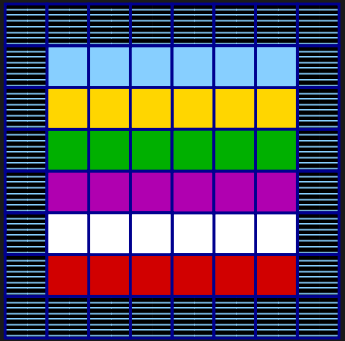




2D diffusion with MPI

How to divide the work in 2d?

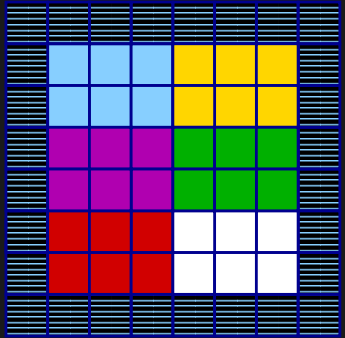




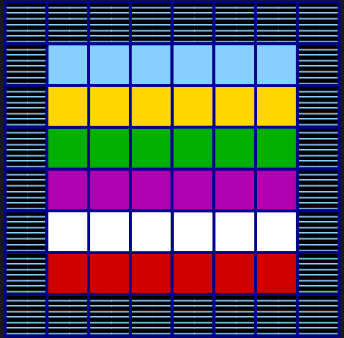
- Less communication (18 edges).
- Harder to program, non-contiguous data to send, left, right, up and down.

2D diffusion with MPI

How to divide the work in 2d?



- Less communication (18 edges).
- Harder to program, non-contiguous data to send, left, right, up and down.

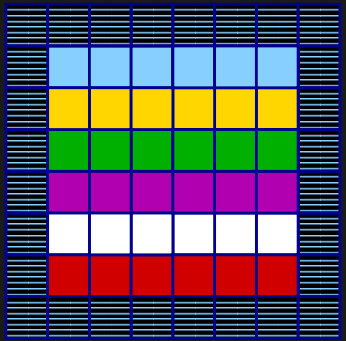


- Easier to code, similar to 1d, but with contiguous guard cells to send up and down.
- More communication (30 edges).

Let's look at the easiest domain decomposition.

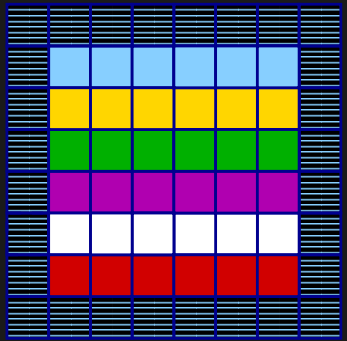
Let's look at the easiest domain decomposition.

Serial:

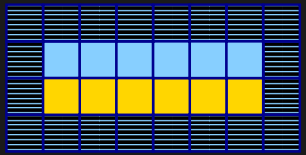


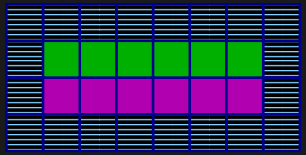
Let's look at the easiest domain decomposition.

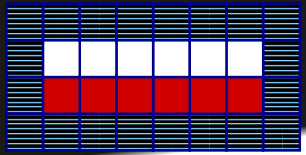
Serial:



Parallel ($P = 3$)*:*

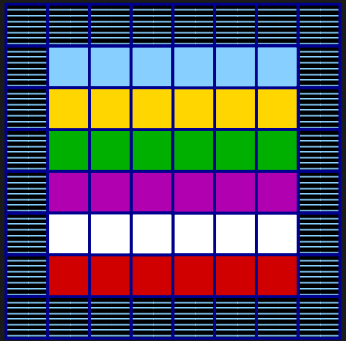




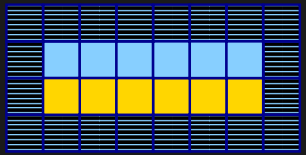


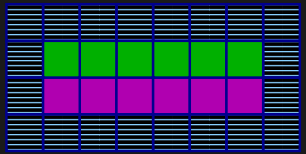
Let's look at the easiest domain decomposition.

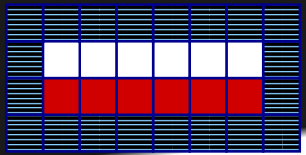
Serial:



Parallel $(P = 3)$***:***







Communication pattern:

- Copy upper stripe to upper neighbour bottom guard cell.
- Copy lower stripe to lower neighbout top guard cell.
- Contiguous cells: can use `count` in `MPI_Sendrecv`.
- Similar to 1d diffusion.

Hands-on: 1D MPI Diffusion

- Serial code:

```
$ cd $SCRATCH/2_mpi/diffusion
$ `# source ../setup
$ make diffusionc # or diffusionf
$ ./diffusionc
```

Hands-on: 1D MPI Diffusion

- Serial code:

```
$ cd $SCRATCH/2_mpi/diffusion
$ `# source ../setup
$ make diffusionc # or diffusionf
$ ./diffusionc
```

- `cp diffusion.c diffusionc-mpi.c`
  or
  `cp diffusion.f90 diffusionf-mpi.f90`
- Make an MPI-ed version!
- Build with `make diffusionc-mpi` or `make diffusionf-mpi`.
- Test on 1..8 processors

Hands-on: 1D MPI Diffusion

- Serial code:

```
$ cd $SCRATCH/2_mpi/diffusion
$ `# source ../setup
$ make diffusionc # or diffusionf
$ ./diffusionc
```

- `cp diffusion.c diffusionc-mpi.c`
  or
  `cp diffusion.f90 diffusionf-mpi.f90`
- Make an MPI-ed version!
- Build with `make diffusionc-mpi` or `make diffusionf-mpi`.
- Test on 1..8 processors

Plan of Attack

- Switch off graphics (in Makefile, change `USEPGPLOT=-DPGPLOT` to `USEPGPLOT=`);
- Add standard MPI calls: init, finalize, comm_size, comm_rank;
- Figure out how many points each process is responsible for (~totpoints/size);
- Figure out neighbors;
- Start at 1, but end at totpoints/size;
- At end of step, exchange guardcells; use sendrecv;
- Get total error.

MPI Summary

MPI Summary - C syntax

```
MPI_Status status;

err = MPI_Init(&argc, &argv);

err = MPI_Comm_{size,rank}(Communicator, &{size,rank});

err = MPI_Send(sendptr, count, MPI_TYPE, destination, tag, Communicator);

err = MPI_Recv(rcvptr, count, MPI_TYPE, source, tag, Communicator, &status);

err = MPI_Sendrecv(sendptr, count, MPI_TYPE, destination,tag, recvptr, count, MPI_TYPE, source, tag, Comm

err = MPI_Allreduce(&mydata, &globaldata, count, MPI_TYPE, MPI_OP, Communicator);

Communicator -> MPI_COMM_WORLD
MPI_Type -> MPI_FLOAT, MPI_DOUBLE, MPI_INT, MPI_CHAR...
MPI_OP -> MPI_SUM, MPI_MIN, MPI_MAX,...
```

MPI Summary - FORTRAN syntax

```
integer status(MPI_STATUS_SIZE)

call MPI_INIT(err)

call MPI_COMM_{SIZE,RANK}(Communicator, {size,rank},err)

call MPI_SSEND(sendarr, count, MPI_TYPE, destination, tag, Communicator)

call MPI_RECV(rcvarr, count, MPI_TYPE, destination,tag, Communicator, status, err)

call MPI_SENDRECV(sendptr, count, MPI_TYPE, destination,tag, recvptr, count, MPI_TYPE, source, tag, Commu

call MPI_ALLREDUCE(mydata, globaldata, count, MPI_TYPE, MPI_OP, Communicator, err)

Communicator -> MPI_COMM_WORLD
MPI_Type -> MPI_REAL, MPI_DOUBLE_PRECISION, MPI_INTEGER, MPI_CHARACTER
MPI_OP -> MPI_SUM, MPI_MIN, MPI_MAX,...
```

Non-blocking communications

MPI Non-Blocking Communications

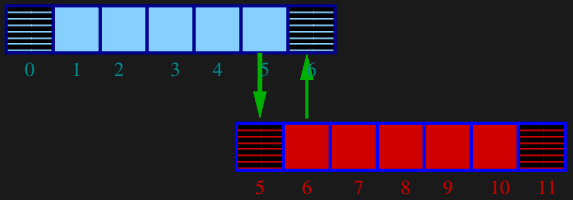
- Mechanism for overlapping/interleaving communications and useful computations
- Avoid deadlocks
- Can avoid system buffering, memory-to-memory copying and improve performance

MPI Non-Blocking Functions: `MPI_Isend`, `MPI_Irecv`

- Returns immediately, posting request to system to initiate communication.
- However, communication is not completed yet.
- Cannot tamper with the memory provided in these calls until the communication is completed.

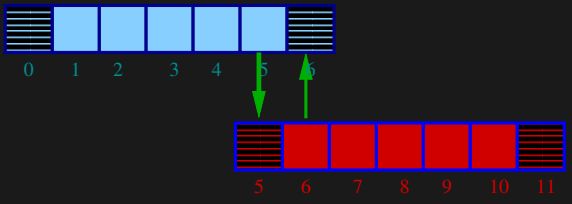
Diffusion: Had to wait for communications to compute

- Could not compute end points without guardcell data
- All work halted while all communications occurred
- Significant parallel overhead.

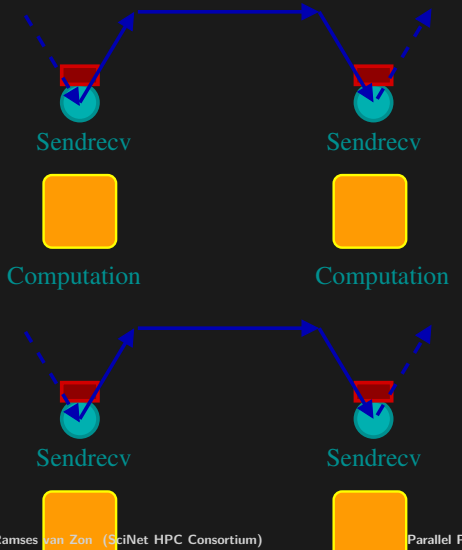


Diffusion: *Had* to wait?

- But inner zones could have been computed just fine.
- Ideally, would do inner zones work while communications is being done; then go back and do end points.



Blocking Communication/Computation Pattern

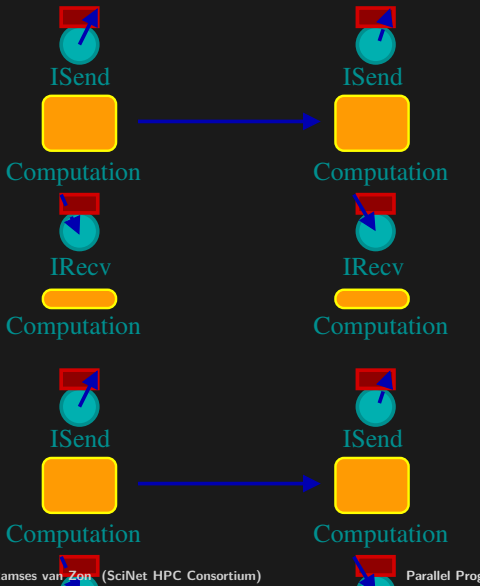


We have the following sequence of communication and computation:

- The code exchanges guard cells using `Sendrecv`
- The code **then** computes the next step.
- The code exchanges guard cells using `Sendrecv` again.
- etc.

We can do better.

Non-Blocking Communication/Computation Pattern



- The code start a send of its guard cells using `ISend`.
- Without waiting for that send's completion, the code computes the next step for the inner cells (while the guard cell message is *in flight*)
- The code then receives the guard cells using `IRecv`.
- Afterwards, it computes the outer cell's new values.
- Repeat.

Nonblocking Sends

- Allows you to get work done while message is 'in flight'
- Must not alter send buffer until send has completed.
- C:

```
MPI_Isend(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *req
```

- FORTRAN:

```
MPI_ISEND(BUF,INTEGER COUNT,INTEGER DATATYPE,INTEGER DEST,INTEGER TAG, INTEGER COMM, INTEGER REQUEST,INTE
```

MPI: Non-Blocking Isend & Irecv

```
err = MPI_Isend(sendptr, count, MPI_TYPE, destination,tag, Communicator, MPI_Request)
err = MPI_Irecv(rcvptr, count, MPI_TYPE, source, tag,Communicator, MPI_Request)
```

- sendptr/rcvptr: pointer to message
- count: number of elements in ptr
- MPI_TYPE: one of MPI_DOUBLE, MPI_FLOAT, MPI_INT, MPI_CHAR, etc.
- destination/source: rank of sender/receiver
- tag: unique id for message pair
- Communicator: MPI_COMM WORLD or user created
- MPI_Request: Identify comm operations

How to tell if message is completed?

- `int MPI_Wait(MPI_Request *request,MPI_Status *status);`
- `MPI_WAIT(INTEGER REQUEST,INTEGER STATUS(MPI_STATUS_SIZE),INTEGER ERROR)`
- `int MPI_Waitall(int count,MPI_Request *array_of_requests, MPI_Status *array_of_statuses);`
- `MPI_WAITALL(INTEGER COUNT,INTEGER ARRAY_OF_REQUESTS(*),INTEGER ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*),INTEGER`

Also: `MPI_Waitany`, `MPI_Test` ...

MPI: Wait & Waitall

Will block until the communication(s) complete

```
err = MPI_Wait(MPI_Request *, MPI_Status *)
err = MPI_Waitall(count, MPI_Request *, MPI_Status*)
```

- `MPI_Request`: Identify comm operation(s)
- `MPI_Status`: Status of comm operation(s)
- count: Number of comm operations(s)

MPI: Test

- Does not block, returns immediately
- Provides another mechanism for overlapping communication and computation.

```
err = MPI_Test(MPI_Request *, flag, MPI_Status *)
```

- `MPI_Request`: Identify comm operation(s)
- `MPI_Status`: Status of comm operation(s)
- `flag`: true if comm complete; false if not sent/recv yet

Hands On

- In diffusion directory, cp diffusion{c,f}-mpi.{c,f90} to diffusion{c,f}-mpi-nonblocking.{c,f90}
- Change to do non-blocking IO; post sends/recvs, do inner work, wait for messages to clear, do end points

MPI-IO

MPI-IO

- Would like I/O to be parallel and not serial
- But writing one file per process is inconvenient and inefficient.
- MPI-IO = The parallel I/O part of the MPI-2 standard.
- Many other parallel I/O solutions are built upon it.
- Versatile and better performance than standard unix IO.
- Usually collective I/O is the most efficient.

MPI-IO exploits analogies with MPI

- Writing $\leftrightarrow$ Sending message
- Reading $\leftrightarrow$ Receiving message
- File access grouped via communicator: collective operations
- User defined MPI datatypes for e.g. non-contiguous data layout
- IO latency hiding much like communication latency hiding
  (IO may even share network with communication)
- All functionality through function calls.

Basic IO Operations (C)

```
int MPI_File_open(MPI_Comm comm, char*filename, int amode, MPI_Info info, MPI_File* fh)

int MPI_File_seek(MPI_File fh, MPI_Offset offset, int to)

int MPI_File_set_view(MPI_File fh,MPI_Offset disp,MPI_Datatype etype,  MPI_Datatype filetype,
char* datarep, MPI_Info info)

int MPI_File_read(MPI_File fh, void* buf, int count,  MPI_Datatype datatype, MPI_Status*status)

int MPI_File_write(MPI_File fh, void* buf, int count, MPI_Datatype datatype, MPI_Status*status)

int MPI_File_close(MPI_File* fh)
```

Basic IO Operations (Fortran)

```
MPI_FILE_OPEN(comm,filename,amode,info,fh,err)
character*(*) filename
integer comm,amode,info,fh,err

MPI_FILE_SEEK(fh,offset,whence,err)
integer(kind=MPI_OFFSET_KIND) offset
integer fh,whence,err

MPI_FILE_SET_VIEW(fh,disp,etype,filetype,datarep,info,err)
integer(kind=MPI_OFFSET_KIND) disp
integer fh,etype,filetype,info,err
character*(*) datarep

MPI_FILE_READ(fh,buf,count,datatype,status,err)
<type> buf(*)
integer fh,count,datatype,status(MPI_STATUS_SIZE),err

MPI_FILE_WRITE(fh,buf,count,datatype,status,err)
<type> buf(*)
integer fh,count,datatype,status(MPI_STATUS_SIZE),err

MPI_FILE_CLOSE(fh)
integer fh
```

Opening and closing a file

As in regular I/O, files are maintained through file handles. A file gets opened with MPI_File_open. E.g. the following codes open a file for reading, and close it right away:

in C:

```
MPI_FILE fh;
MPI_File_open(MPI_COMM_WORLD,"test.dat",MPI_MODE_RDONLY,MPI_INFO_NULL,&fh);
MPI_File_close(&fh);
```

in Fortran:

```
integer :: fh, err
call MPI_FILE_OPEN(MPI_COMM_WORLD,"test.dat",MPI_MODE_RDONLY,MPI_INFO_NULL,fh,err)
call MPI_FILE_CLOSE(fh,err)
```

Opening a file requires...

- communicator,
- file name,
- file handle, for all future reference to file,
- info structure, or `MPI_INFO_NULL`,
- file mode, made up of combinations of the following

Opening a file requires. . .

- communicator,
- file name,
- file handle, for all future reference to file,
- info structure, or `MPI_INFO_NULL`,
- file mode, made up of combinations of the following

- `MPI_MODE_RDONLY`: read only
- `MPI_MODE_RDWR`: reading and writing
- `MPI_MODE_WRONLY`: write only
- `MPI_MODE_CREATE`: create the file if it does not exist
- `MPI_MODE_EXCL`: error if creating a file that exists
- `MPI_MODE_DELETE_ON_CLOSE`: delete file on close
- `MPI_MODE_UNIQUE_OPEN`: file not to be opened elsewhere
- `MPI_MODE_SEQUENTIAL`: file to be accessed sequentially
- `MPI_MODE_APPEND`: position all file pointers to

etypes, filetypes, file views

To make binary access a bit more natural for many applications, MPI-IO defines file access through the following concepts:

- displacement: Where to start in the file.
- etype: Allows to access the file in units other than bytes.
- filetype: Each process defines what part of a shared file it uses.
  - Filetypes specify a pattern which gets repeated in the file.
  - Useful for non-contiguous access.
  - For contiguous access, often etype=filetype.

Together, these three specify the file view.
File views have to be defined collectively with `MPI_File_set_view`.
If no view is defined, a default view is active, with etype `MPI_BYTE`, and displacement 0.

Overview of all read functions

.	**Single task**	**Collective**
Individual file pointer	.	.
blocking	MPI_File_read	MPI_File_read_all
nonblocking	MPI_File_iread	MPI_File_read_all_begin
.	+(MPI_Wait)	MPI_File_read_all_end
Explicit offset	.	.
blocking	MPI_File_read_at	MPI_File_read_at_all
nonblocking	MPI_File_iread_at	MPI_File_read_at_all_begin
.	+(MPI_Wait)	MPI_File_read_at_all_end
Shared file pointer	.	.
blocking	MPI_File_read_shared	MPI_File_read_ordered
nonblocking	MPI_File_iread_shared	MPI_File_read_ordered_begin
.	+(MPI_Wait)	MPI_File_read_ordered_end

Overview of all write functions

.	**Single task**	**Collective**
Individual file pointer	.	.
blocking	MPI_File_write	MPI_File_write_all
nonblocking	MPI_File_iwrite	MPI_File_write_all_begin
.	+(MPI_Wait)	MPI_File_write_all_end
Explicit offset	.	.
blocking	MPI_File_write_at	MPI_File_write_at_all
nonblocking	MPI_File_iwrite_at	MPI_File_write_at_all_begin
.	+(MPI_Wait)	MPI_File_write_at_all_end
Shared file pointer	.	.
blocking	MPI_File_write_shared	MPI_File_write_ordered
nonblocking	MPI_File_iwrite_shared	MPI_File_write_ordered_begin
.	+(MPI_Wait)	MPI_File_write_ordered_end

Choices

Collective?

After a file has been opened and a fileview is defined in each process, processes can independently read and write to their part of the file.
But if the IO occurs at regular spots in the program, which different processes reach the same time, it will be better to use collective I/O.
These are the `_all` versions of the MPI-IO routines.

Two file pointers

An MPI-IO file has two different file pointers:

- individual file pointer: one per process.
- shared file pointer: one per file: `_shared/_ordered`

"Shared" doesn't mean "collective", but does imply synchronization!

Choices

Strategic considerations

Pros for single task I/O:

- One can virtually always use only indivivual file pointers,
- If timings variable, no need to wait for other processes

Cons:

- If there are interdependences between how processes write, there may be collective I/O operations may be faster.
- Collective I/O can collect data before doing the write or read.

True speed depends on file system, size of data to write and implementation.

Non-contiguous data

What if the data in the file is supposed to be as follows?

- Filetypes can help!
- Or custom MPI data types (also useful in high dimensional ghost cells).

Overview of data/filetype constructors

Function	Creates a ...
`MPI_Type_contiguous`	contiguous datatype
`MPI_Type_vector`	vector (strided) datatype
`MPI_Type_indexed`	indexed datatype
`MPI_Type_indexed_block`	indexed datatype w/uniform block length
`MPI_Type_create_struct`	structured datatype
`MPI_Type_create_resized`	type with new extent and bounds
`MPI_Type_create_darray`	distributed array datatype
`MPI_Type_create_subarray`	n-dim subarray of an n-dim array
...	...

Before using the create type, you have to do `MPI_Commit`.

File data representation

There are three possible representations:

- native:
  Data is stored in the file as it is in memory: no conversion is performed. No loss in performance, but not portable.
- internal:
  Implementation dependent conversion. Portable across machines with the same MPI implementation, but not across different implementations.
- external32:
  Specific data representation, basically 32-bit big-endian IEEE format.

See MPI Standard for more info. Completely portable, but not the best performance.
These have to be given to `MPI_File_set_view` as strings.

More non-contiguous data: subarrays

What if there's a large 2d matrix that is distributed across processes?
Common special cases of non-contiguous access → specialized functions: `MPI_File_create_subarray` and `MPI_File_create_darray`.
C code:

```
int gsizes[2]={16,6};
int lsizes[2]={8,3};
int psizes[2]={2,2};
int coords[2]={rank%psizes}[0],rank/psizes[0]};
int starts[2]={coords}[0]*lsizes}[0],coords[1]*lsizes[1]};
MPI_Type_create_subarray(2,gsizes,lsizes,starts,MPI_ORDER_C,MPI_INT,&filetype);
MPI_Type_commit(&filetype);
MPI_File_set_view(fh},0,MPI_INT,filetype,"native",MPI_INFO_NULL);
MPI_File_write_all(fh,local_array,local_array_size,MPI_INT,MPI_STATUS_IGNORE);
```

Tip: `MPI_Cart_create` can be useful to compute coordinatess for a process.